Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño e implementación de un analizador de biosecuencias. Análisis de subsecuencias de repetición

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Villar Lafuente, Carlos José

Tutor: Sempere Luna, José María

2015/2016

A todos aquellos que me han apoyado en los momentos difíciles, gracias.

Resumen

En el presente trabajo fin de grado (TFG) se desarrolla un analizador de secuencias de carácter bioquímico, fundamentalmente, de secuencias de ácido desoxirribonucleico (ADN) y de ácido ribonucleico (ARN), para la búsqueda de posibles mutaciones debidas a la repetición de subsecuencias.

El marco teórico de este trabajo se basa en la utilización de lenguajes formales (incontextuales y regulares) y de algoritmos de análisis eficientes para dichos lenguajes. El analizador se desarrolla en el lenguaje de programación JAVA, con él se pretende obtener una herramienta que trabaje en modo *stand-alone*. Posteriormente, se evaluará su posible integración en un sitio web para su ejecución en red.

Palabras clave: Bioinformática, lenguajes formales, gramáticas y autómatas, análisis de secuencias.

Abstract

In the present project, a sequence analyzer biochemical character develops mainly of sequences of deoxyribonucleic acid (DNA) and ribonucleic acid (RNA), to search for possible mutations due to repetition of subsequences.

The theoretical framework of this paper is based on the use of formal languages (context-free and regular) and analysis of efficient algorithms for these languages. The analyzer is developed in the Java programming language, with it is to obtain a tool that works in stand- alone mode. Subsequently, possible integration will be evaluated on a website to run on network.

Keywords: Bioinformatics, formal languages , grammars and automata, sequence analysis.

Tabla de contenidos

1. Introducción

Los seres vivos estamos formados por millones de células, la célula es la unidad viva con menor tamaño, es capaz de actuar autónomamente, es decir, puede realizar por sí misma las funciones de reproducción, nutrición y relación. Todas las células eucariotas están compuestas de la misma estructura, poseen una membrana, un citoplasma y un núcleo. Dentro del núcleo existe una serie de cadenas de aminoácidos que codifican toda la información necesaria para "construir" el ser vivo. Dichas cadenas de aminoácidos son cadenas de ácido desoxirribonucleico (ADN) y ácido ribonucleico (ARN).

Es sabido por todos que las cadenas de ADN y ARN están representadas por una sucesión de símbolos (G, A, T y C en el caso del ADN y G, A, U y C en el caso del ARN), que son replicadas una y otra vez y que en el proceso se producen "errores" de duplicado (conocidos como mutaciones).

Un ejemplo de mutación, y objetivo de estudio del TFG, es la mutación por repetición. En el proceso de duplicación, un fragmento de la cadena es repetido a continuación de sí mismo, es decir GATTACA se puede convertir en GATTATACA.

En este contexto aplicamos el conocimiento adquirido sobre autómatas, gramáticas incontextuales y lenguajes formales, mediante los cuales podemos crear un lenguaje formado por las cadenas resultantes de duplicar subsecuencias de la cadena original, de forma iterada, su gramática generadora y por lo tanto averiguar si una cadena es producto de otra tras una serie de mutaciones.

Para ello se ha creado un programa en JAVA que, dada una cadena de entrada, obtiene una gramática y con esta evalúa si una cadena de test es el resultado de varias mutaciones de la primera.

2. Conceptos básicos

2.1. Biología celular y bioquímica

Tal y cómo se ha mencionado en la introducción, la célula es la unidad viva con menor tamaño, es capaz de actuar autónomamente, esto es que puede reproducirse, nutrirse y relacionarse por sí misma. Según el número de células que conforman el ser vivo, este puede clasificarse como unicelular (una sola célula) o pluricelular (conjunto de células que desarrollan funciones distintas para el ser vivo), este último puede ser de cientos de células a billones de estas.

Todas las células eucariotas (aquellas que son objeto del estudio, ya que las eucariotas de tipo animal son las que mediante las cuales el ser humano está constituido) están rodeadas por una envoltura de tipo membrana que las separa y comunica con el exterior (conocida como membrana plasmática). Dentro se encuentra el citoplasma que constituye la mayor parte del fluido intracelular en las células eucariotas. Finalmente se encuentra el núcleo, hogar del ADN, donde están los cromosomas (unidades que contienen el ADN agrupado), que contienen toda la información de cómo se debe constituir el ser vivo, desde la célula en sí a todo el compendio de células que es el ser humano. Esto no quiere decir que sean las únicas partes de una célula, en realidad existen muchas más, pero sí que son las principales.

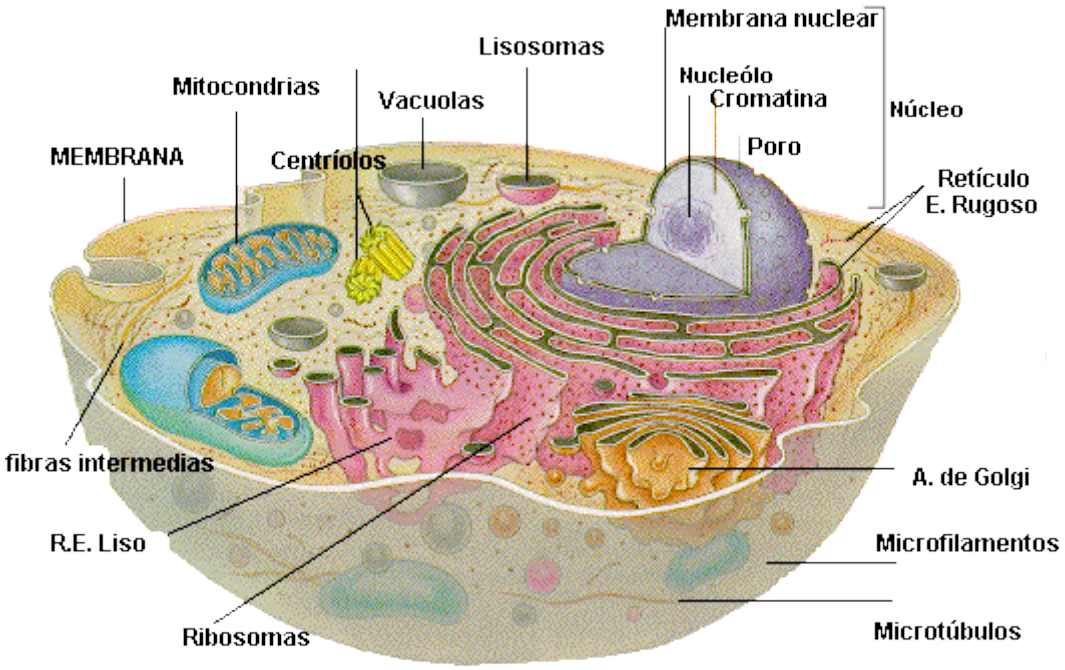

Fig. 1 - Partes de la célula.

Como podemos observar en la figura 1, la célula está compuesta de muchas partes, pero aquella que nos interesa es el núcleo, donde se produce el proceso de duplicación del ADN. El proceso de duplicación del ADN lo veremos más adelante.

Llamamos bioquímica a la rama de la ciencia encargada del estudio de los procesos biológicos y funciones de los seres vivos desde el punto de vista de la química orgánica. Estudiando las sustancias que se encuentran en los seres vivos y sus reacciones químicas fundamentales en los procesos vitales.

El campo de estudio de la bioquímica es especialmente amplio, ya que busca comprender a nivel molecular, químicamente, procesos tales como la reproducción, la alimentación, el funcionamiento de nuestro sistema nervioso, etc...Pero en el caso que nos atañe es el del ADN y el ARN.

El ADN y el ARN son ácidos nucleicos, que son polímeros formados por monómeros denominados nucleótidos (ver figura 2) y unidos por enlaces fosfodiéster [1], formando largas cadenas que almacenan información genética sobre organismos vivos.

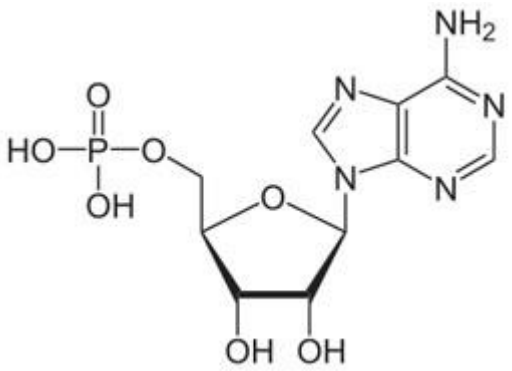

Fig. 2 - Nucleótido

Los enlaces fosfodiéster son un tipo de enlace covalente (enlace entre dos átomos que se produce cuando estos átomos se unen, para alcanzar el octeto estable, compartiendo electrones del último nivel) que se produce entre un grupo hidroxilo (grupo funcional formado por un átomo de oxígeno y otro de hidrógeno, característico de los alcoholes, fenoles y ácidos carboxílicos entre otros) en el carbono 3’ y un grupo fosfato (que está compuesto por un átomo central de fósforo rodeado por cuatro átomos idénticos de oxígeno en disposición tetraédrica) en el carbono 5' del nucleótido entrante, formándose así un doble enlace éster. En esta reacción se libera una molécula de agua y se forma un dinucleótido.

En el ADN, las cadenas se encuentran en forma de doble hélice formando pares de monómeros (o bases) complementarias entre sí. Dichos nucleótidos están formados por un azúcar (la desoxirribosa en el caso del ADN y ribosa en el del ARN), un fosfato y una base (Adenina, Citosina, Guanina y Timina en el ADN o Uracilo en el del ARN).

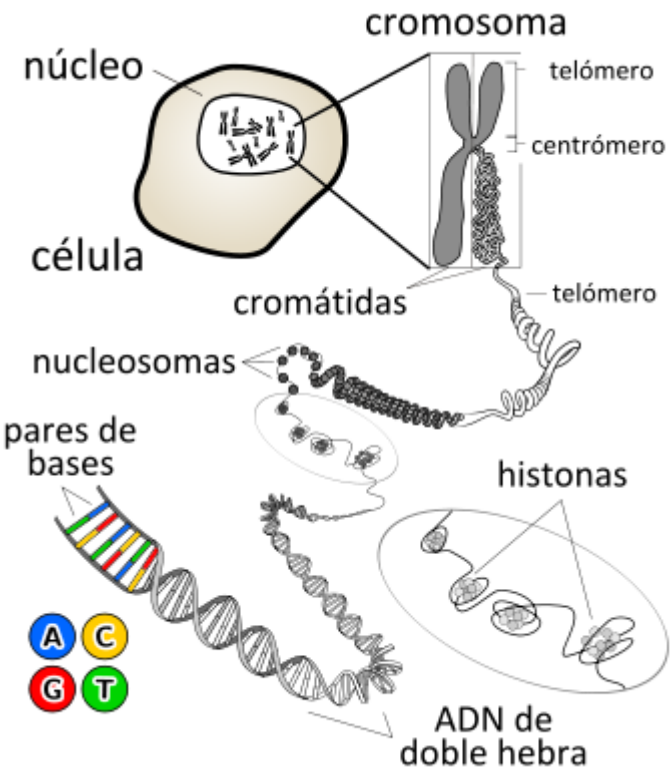

Fig. 3 - Situación del ADN dentro de una célula eucariota. KES47 (2010) Descomposición gráfica de un cromosoma (encontrado en el núcleo de la célula), hasta los pares de bases del ADN. Recuperado de Wikipedia

En el proceso de duplicación del ADN la doble hélice se abre como una cremallera, separando los pares de bases por el puente de hidrógeno que los une y posteriormente cada hebra sirve de molde para la creación de dos nuevas dobles hélices uniendo proteínas a las hebras.

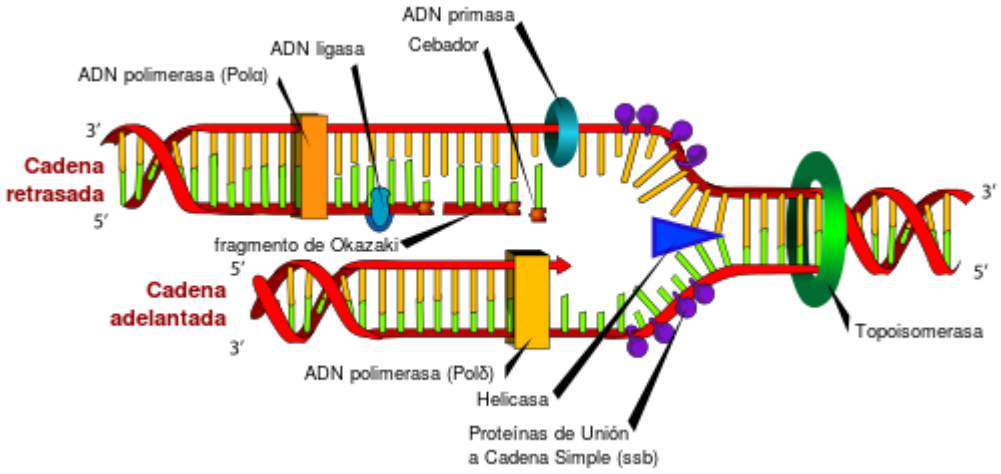

Fig. 4 - Proceso de replicación del ADN

Este proceso no es perfecto, durante la duplicación y reconstrucción se pueden producir errores y obtener una cadena distinta a la original. Estos errores se conocen como mutaciones y son los responsables, entre otras cosas, de la adaptación de los seres vivos al entorno (cuando la mutación se produce en una célula reproductora y ocasiona una característica que le permite al ser vivo una mejor supervivencia al medio y mejora sus posibilidades de reproducción).

Una de las más frecuentes mutaciones (y a la vez menos entendida) es la de duplicación de segmentos, es decir, dada una cadena de símbolos xyz pasa a ser xyyz. Durante la duplicación se producen varias copias de la hebra de ADN y estas pueden cruzarse, provocando borrado o duplicación de subsecuencias. A continuación, se observa como dada dos veces la misma cadena w, se cruzan entre sí, dando lugar a la repetición de secuencias en el proceso.

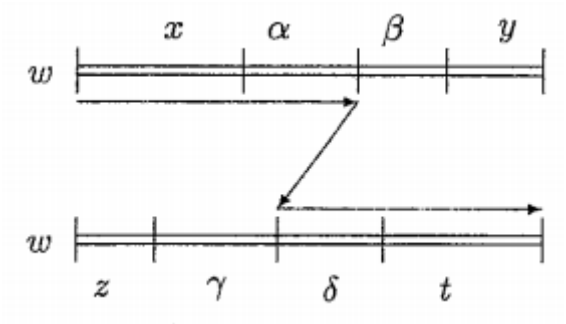

Fig. 5 - Esquema de la duplicación en el genoma

2.2. Teoría de lenguajes, gramáticas y autómatas

Primero definimos un alfabeto como un conjunto finito no vacío de elementos llamados símbolos. Una cadena (o palabra) sobre dicho alfabeto es una sucesión finita de los símbolos encontrados en el alfabeto. Sea por ejemplo el alfabeto $A = \{a, b, c\}$, una palabra sobre A sería por ejemplo $abbc$. Denotamos mediante ε la palabra vacía, es decir, que su longitud (el número de símbolos que forma la cadena, $|\varepsilon|$) es cero. Mediante $|x|_a$ hacemos referencia al número de veces que aparece el símbolo a en la cadena x.

Definimos A^* como el conjunto con todas las palabras posibles con los símbolos del alfabeto A, es decir, $A^* = \{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, ...\}$. Definimos un lenguaje como un subconjunto de A^* (este subconjunto puede ser finito o no). Definimos A^+ al subconjunto formado por $A^* - \{\varepsilon\}$.

Así pues, un lenguaje formal es un subconjunto de A^* donde las palabras son formadas acorde a unas reglas formalmente definidas (mediante una estructura matemática), dichas reglas se conocen como gramática formal.

El homomorfismo se define como una función entre grupos que preserva la operación binaria (operación que requiere dos operandos). En grupos la utilizamos para realizar un cambio de dominio.

Una gramática formal G se define como una tupla (N, T, P, S), donde N y T son los alfabetos de símbolos auxiliares y terminales respectivamente, siendo la intersección de ambos alfabetos el conjunto vacío. Los símbolos auxiliares se representan con mayúsculas y con minúsculas los terminales. S es el símbolo inicial de la gramática (perteneciente a N). P es el conjunto finito de reglas (o producciones) de la forma α->β (se lee α produce β).

Partiendo del símbolo S, se suceden una serie de transformaciones arbitrarias de símbolos según las producciones definidas obteniendo como resultado una cadena de símbolos terminales, perteneciente al lenguaje L definido mediante G.

Ejemplo 1. Sea la gramática definida por las producciones

S -> aBSc | abc

Ba -> aB

Bb -> bb

B -> b

Partiendo de S una serie de transformaciones (o derivaciones) sería:

S -> aBSc -> aBabcc -> aaBbcc -> aabbcc

Con derivación hacemos referencia a cada vez que transformamos la parte izquierda de una producción a la derecha (con reducción hacemos referencia al paso inverso). Denotamos con L(G) como el lenguaje definido por la gramática *G*, es decir, las palabras que se pueden generar con el conjunto de reglas definido.

Noam Chomsky definió una jerarquía de gramáticas basadas en la forma que adoptan las producciones, esta jerarquía se divide en cuatros grupos:

0. Gramáticas no restringidas: No tienen ningún tipo de restricción
1. Gramáticas sensibles al contexto: Las producciones toman la forma de $\alpha A\beta \rightarrow \alpha\gamma\beta$ $\alpha, \beta \in (N \cup T)^*$, $A \in N$, $\gamma \in (N \cup T)^+$. También se permite la regla $S \rightarrow \varepsilon$ siempre que S no aparezca como consecuente de ninguna producción
2. Gramáticas incontextuales: Las producciones toman la forma de $A \rightarrow \alpha$ $\alpha \in (N \cup T)^*$, $A \in N$
3. Gramáticas regulares por la izquierda (o derecha): Las producciones toman forma de:

Gramática lineal por la izquierda	Gramática lineal por la derecha
$A \rightarrow Ba$ $a \in T, A, B \in N$	$A \rightarrow aB$ $a \in T, A, B \in N$
$A \rightarrow a$ $a \in T, A \in N$	$A \rightarrow a$ $a \in T, A \in N$
$A \rightarrow B$ $A, B \in N$	$A \rightarrow B$ $A, B \in N$
$A \rightarrow \lambda$ $A \in N$	$A \rightarrow \lambda$ $A \in N$

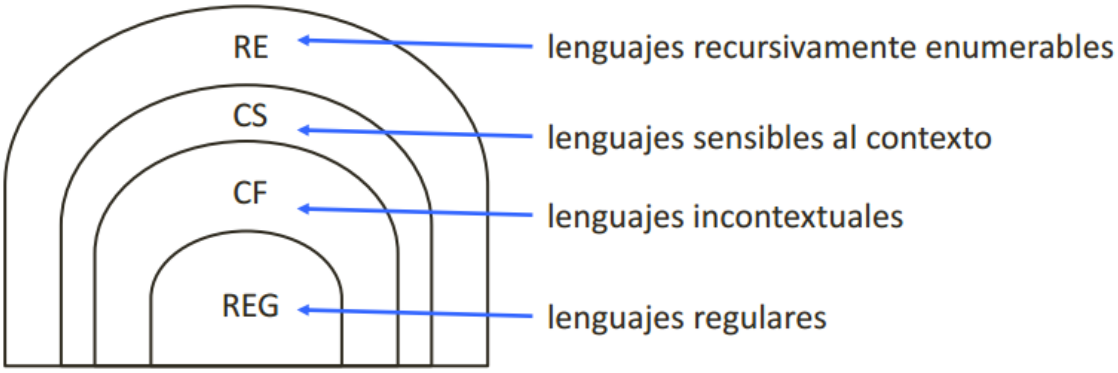

Fig. 6 - Representación jerárquica de los lenguajes según Chomsky

Decimos que una gramática incontextual está en forma normal de Chomsky cuando sus producciones toman una de las siguientes formas:

$A \rightarrow BC$ $A, B, C \in N$

$A \rightarrow a$ $A \in N, a \in T$

Un autómata es una máquina de estados con un control finito que ante una cadena de entrada y siguiendo una función de transición, determina a qué estado cambia según el estado actual y el símbolo consumido por la cinta de entrada. Existen varios.

Un autómata finito es aquel que consiste en un conjunto finito de estados y un conjunto de transiciones de estado a estado que ocurren con la lectura de la entrada de un símbolo del alfabeto ∑. Por cada símbolo hay exactamente una transición desde cada uno de los estados que puede acabar en cualquier estado (incluido el de origen), en el caso determinista. Algunos estados son designados como finales o estados de aceptación. Se asocia un grafo dirigido, llamado diagrama de transición, a un autómata finito, siendo los vértices los estados y las aristas las transiciones del estado q a p con el símbolo a. El autómata acepta una cadena si tras procesar todos los símbolos termina en un estado designado como final, partiendo del estado inicial.

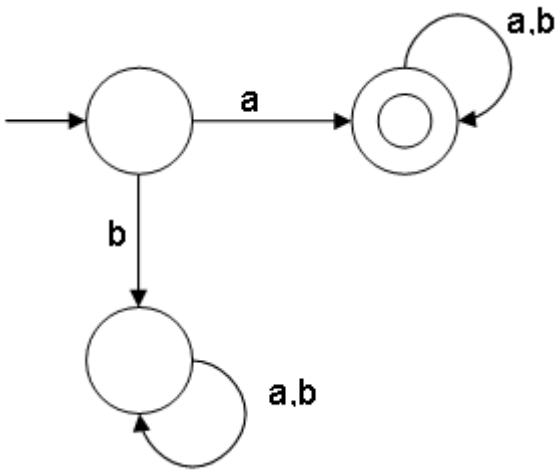

Fig. 7 - Diagrama de transición de un autómata finito

Al igual que la gramática, los autómatas tienen un lenguaje asociado, en este caso, el autómata A (el de la figura 7) reconoce cadenas de a y b que empiezan por a, o lo que es lo mismo, L(A) = {a, aa, ab, aaa, aab, aba, abb, aaaa, ...}.

Una máquina de Turing se representa como una máquina con una cinta infinita en la que puede leer y escribir símbolos (puede tener más de una cinta sobre la que operar), dispone de un cabezal que le permite leer/escribir en cada una de las celdas de la cinta y puede desplazarse sobre esta. Dispone de un control finito mediante el cual, siguiendo una función de transición en función del estado en el que se encuentre y el símbolo de la cinta que lee el cabezal, se cambia de estado, se escribe un símbolo se mueve el cabezal.

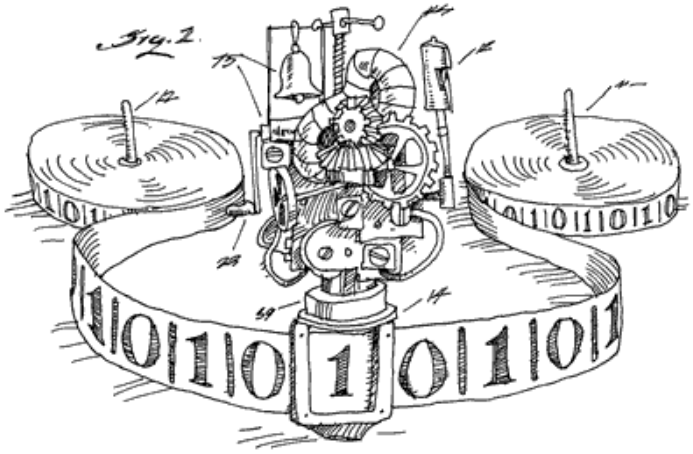

Fig. 8 - Representación artística de una máquina de Turing

Un autómata linealmente acotado sería una máquina de Turing en la que, en lugar de tener una cinta infinita, tenemos una cinta acotada a longitud n, pues el lenguaje aceptado por dicho autómata sería aquel cuyas palabras cupiesen en la cinta de entrada y el autómata tras procesarla parase y aceptase la palabra introducida.

Un autómata de pila [6] puede ser visto como un autómata finito, con una cinta de entrada, un control de estado y una pila. La pila es una cadena de símbolos donde el símbolo más a la izquierda representa el que está en lo alto de la pila. Puede ser no determinista, teniendo un conjunto finito de posibles acciones en cada momento de la computación.

Así pues, el autómata de pila lo definimos como sigue:

$$M = (Q, \sum, \Gamma, \delta, q_o, Z_o, F)$$

1) Q es el conjunto de estados;
2) $\sum$ es el alfabeto de entrada;
3) Γ es el alfabeto de pila;
4) q_o, perteneciente a Q, es el estado inicial;
5) Z_o, perteneciente a Γ, es un símbolo particular de la pila llamado símbolo de inicio;
6) F es un subconjunto de estados de Q que denominamos estados finales;
7) δ es la función de transición, consiste en un mapeado de $Q \times (\sum \cup \{\varepsilon\}) \times \Gamma$ a un subconjunto finito de $Q \times \Gamma^*$.

El autómata de pila tiene dos posibles movimientos, el primero donde un símbolo de la cadena de entrada será consumido y, dependiendo del estado, símbolo consumido y símbolo en lo alto de la pila, se realizará una de las posibles acciones, transicionando al nuevo estado que indique y manipulando la pila (sustituyendo la cima de esta por una cadena de símbolos indicada).

El otro movimiento se conoce como ε-movimiento, es como el anterior sólo que no es consumido ningún símbolo de la cadena de entrada y el cabezal de lectura no

avanza. Estos movimientos se utilizan para manipular la pila sin consumir símbolos de la cadena de entrada.

Tenemos dos tipos de aceptación en los autómatas de pila: aceptación por pila vacía, que aceptará la palabra de entrada cuando la pila no tenga contenido alguno; y el de aceptación por estado final, que aceptará cuando se detenga en un estado perteneciente al conjunto de los estados finales.

Veamos ahora los movimientos. La interpretación de

$$\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots (p_m, \gamma_m)\}$$

donde q y p_i, $1 \leq i \leq m$, son estados, a pertenece a $\sum$, Z es un símbolo de pila y γ_i, $1 \leq i \leq m$, se encuentra en Γ^*. Cuando el autómata se encuentre en el estado q, haya consumido el símbolo a y en la cima de la pila se encuentre Z, podrá transicionar a uno de los posibles p_i y sustituirá la cima de la pila (Z) por γ_i.

La interpretación de

$$\delta(q, \varepsilon, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots (p_m, \gamma_m)\}$$

es igual que la anterior, pero con la salvedad de que ningún símbolo de la cadena de entrada es consumido, el cabezal no se mueve y da igual lo que esté leyendo.

$M = (\{q_1, q_2\}, \{0, 1\}, \{R, B, G\}, \delta, q_1, R, \varnothing)$

1) $\delta(q_1, 0, R) = \{(q_1, BR)\}$
2) $\delta(q_1, 1, R) = \{(q_1, GR)\}$
3) $\delta(q_1, 0, B) = \{(q_1, BB), (q_2, \epsilon)\}$
4) $\delta(q_1, 0, G) = \{(q_1, BG)\}$
5) $\delta(q_1, 1, B) = \{(q_1, GB)\}$
6) $\delta(q_1, 1, G) = \{(q_1, GG), (q_2, \epsilon)\}$
7) $\delta(q_2, 0, B) = \{(q_2, \epsilon)\}$
8) $\delta(q_2, 1, G) = \{(q_2, \epsilon)\}$
9) $\delta(q_1, \epsilon, R) = \{(q_2, \epsilon)\}$
10) $\delta(q_2, \epsilon, R) = \{(q_2, \epsilon)\}$

Fig. 9 - Autómata no determinista de aceptación por pila vacía

En la figura 9 se observa la definición de un autómata de pila de aceptación por pila vacía que define el lenguaje $\{ww^R \mid w \in (0 + 1)^*\}$. Podemos observar que es no determinista porque los movimientos 3 y 6 tienen dos posibles acciones a realizar. En la siguiente imagen (figura 10) vemos el árbol de derivación (una derivación es cada una de las transiciones de la máquina, cómo deriva de un estado, cadena de entrada y una pila X a un estado, cadena de entrada y una pila Y).

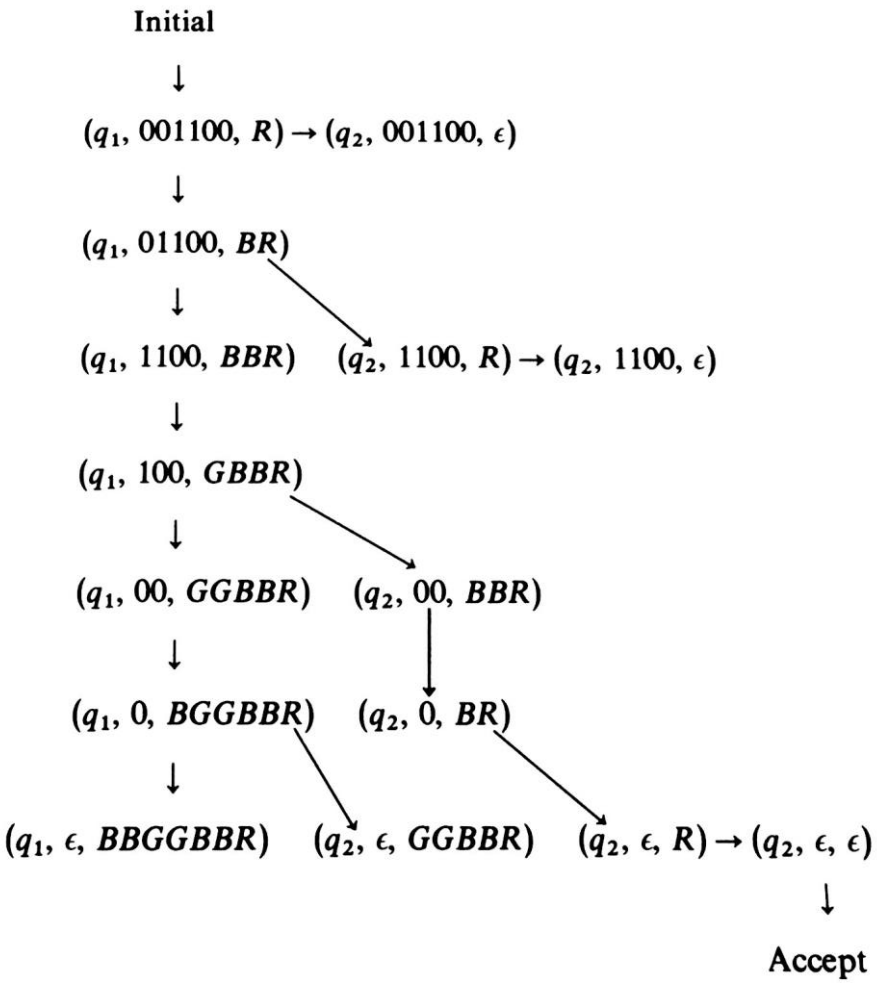

Fig. 10 - Árbol de derivación para el autómata de pila de la figura 9 con la cadena de entrada 001100

Podemos observar en la figura 10 cómo se han producido varias ramificaciones al deberse de un autómata no determinista. Hay que destacar que, aunque la pila se vacíe o la cadena de entrada se consuma por completo, sólo acepta cuando la cadena de entrada ha sido consumida y la pila está vacía.

3. Duplicación de secuencias

Según [2], para un alfabeto $V = \{a_1, a_2, \ldots, a_k\}$ (considerando una ordenación de V), el mapeado de Parikh [3] asociado a V es un homomorfismo Ψ_v desde V^* al monoide del vector de adición sobre $\mathbb{N}^k$, definido por $\Psi_v(s) = (|s|_{a_1}, |s|_{a_2}, \ldots, |s|_{a_k})$, $s \in V^*$; además, dado un lenguaje L sobre V, definimos su imagen a través del mapeado de Parikh como el conjunto $\Psi_v(L) = \{\Psi_v(x) \mid x \in L\}$.

Un subconjunto X de $\mathbb{N}^k$ se dice que es lineal si existen los vectores $c_0, c_1, c_2, \ldots, c_n \in \mathbb{N}^k$, para un $n \geq 0$ tales que $X = \{c_0 + \sum_{i=1}^{n} x_i c_i \mid x_i \in \mathbb{N},\ 1 \leq i \leq n\}$. La unión finita de conjuntos lineales es llamada *semilineal*. Para cualquier entero positivo n, definimos [n] como el conjunto $\{1, 2, 3, \ldots, n\}$.

Sea V un alfabeto y $X \in \{\mathbb{N}\} \cup \{[k] \mid k \geq 1\}$. Para una cadena $w \in V^+$, establecemos el conjunto

$$D_x(w) = \{uxxv \mid w = uxv, u, v \in V^*, x \in V^+, |x| \in X\}.$$

Ahora definimos recursivamente los lenguajes:

$D_X^0(w) = \{w\}$,

$D_X^i(w) = \bigcup_{x \in D_X^{i-1}(w)} D_X(x), i \geq 1$,

$D_X^*(w) = \bigcup_{i \geq 0} D_X^i(w)$.

Los lenguajes $D_{\mathbb{N}}^*(w)$ y $D_{[k]}^*(w)$, $k \geq 1$, son llamados *lenguaje de duplicación no acotado* y *lenguaje de duplicación k-acotado* respectivamente, definidos por w. En otras palabras, para cualquier $X \in \{\mathbb{N}\} \cup \{[k] \mid k \geq 1\}$, $D_X^*(w)$ es el lenguaje $L' \subseteq V^*$ de menor tamaño tal que $w \in L'$ y, para cualquier $uxv \in L'$, $uxxv \in L'$ con $u, v \in V^*$, $x \in V^+$, y $|x| \in X$.

Ahora nos surge la duda de cómo clasificaríamos el lenguaje de duplicación no acotado en la jerarquía de Chomsky. En [4] se observa que el lenguaje de duplicación no acotado definido sobre un alfabeto de más de dos símbolos no es regular, mientras que en [5] muestra que esto es sólo en estos casos en los que el lenguaje es definido a partir de una palabra es no regular. Por lo tanto, el lenguaje de duplicación no acotado definido por una palabra w es regular si y sólo si w contiene dos letras distintas, mientras que si es acotado es regular siempre que el alfabeto no contenga tres o más símbolos si y solo si k es mayor o igual a cuatro.

No sabemos si el lenguaje de duplicación no acotado es libre de contexto o no. Una observación directa conlleva al hecho de que todos estos lenguajes son conjuntos lineales, esto es, la imagen de cada lenguaje de duplicación no acotado a través del mapeado de Parikh es lineal. De hecho, si $w \in V^+$, $V = \{a_1, a_2, \ldots, a_n\}$, entonces uno puede inferir fácilmente que

$$\Psi_v(D_{\mathbb{N}}^*(w)) = \{\Psi_v(w) + \sum_{i=1}^{n} x_i e_i^{(n)} \mid x_i \in \mathbb{N},\ 1 \leq i \leq n\}$$

donde $e_i^{(n)}$ es el vector de tamaño n teniendo en su entrada i-ésima a 1 y el resto de entradas igual a 0.

Dado un lenguaje regular L, uno puede determinar algorítmicamente si L es o no es un lenguaje de duplicación no acotado. Denotemos alph(x) como el alfabeto de menor tamaño posible tal que x $\in$ (alph(x)) *. El algoritmo funciona de a siguiente forma:

1) Encontramos la cadena más corta z perteneciente a L (esto se puede hacer algorítmicamente). Si hay más de una cadena de la longitud de z, entonces L no es un lenguaje de duplicación no acotado.
2) Calculamos la cardinalidad (número de elementos) de alph(z).
3) Si la cardinalidad es mayor o igual a 3, no existe x que cumpla $L = D_{\mathbb{N}}^{*}(x)$.
4) Si la cardinalidad es igual a 1, L es un lenguaje de duplicación no acotado si y solo si $L = \{a^{|z|+m} \mid m \geq 0\}$, y alph(z)={a}.
5) Si k = 2, $z=z_1z_2z_3\ldots z_n$, $z_i \in$ alph(z), $1 \leq i \leq n$, entonces L es un lenguaje de duplicación no acotado si y solo si

$$L=z_1^{+}e_1z_2e_2\ldots e_{n-1}z_n^{+}$$

donde

$$e_i \begin{cases} z_{i+1}^{*}, & si\ z_i = z_{i+1} \\ \{z_i + z_{i+1}\}^{*}, & si\ z_i \neq z_{i+1} \end{cases}$$

para todo $1 \leq i \leq n-1$.

A diferencia de los lenguajes de duplicación no acotados, somos capaces de determinar el lugar que ocupan en la jerarquía de Chomsky los lenguajes de duplicación acotados. Para cualquier palabra r y cualquier entero n ≥ 1, el lenguaje de duplicación n-acotado definido por r es incontextual.

Para nuestro alfabeto V = alph(r) definimos el alfabeto extendido $V_e := V \cup \{\langle a \rangle \mid a \in V\}$. Además, definimos $L^{\leq l} := \{w \in L \mid |w| \leq l\}$ para cada lenguaje L y entero l. Definimos el autómata de pila

$$A_n^r = \left(Q, V, \Gamma, \delta, \begin{bmatrix} \varepsilon \\ \varepsilon \\ r \end{bmatrix}, \bot, \left\{\begin{bmatrix} \varepsilon \\ \varepsilon \\ \varepsilon \end{bmatrix}\right\}\right),$$

donde

$$Q = \left\{ \begin{bmatrix} \mu \\ v \\ w \end{bmatrix} \mid \mu \in (V_e^* \cdot V^* \cup V^* \cdot V_e^*)^{\leq n}, v \in (V^*)^{\leq n}, w \in (V^*)^{\leq |r|} \right\},$$

$$\Gamma = \{\bot\} \cup \left\{ \overline{\mu} \mid \begin{bmatrix} \mu \\ v \\ w \end{bmatrix} \in Q, v \in (V^*)^{\leq n}, \overline{w} \in (V^*)^{\leq |r|} \right\}.$$

Llamamos a las cadenas que hay en cada estado como patrón, memoria y conjetura respectivamente de abajo a arriba. Definimos ahora una función de transición determinista intermedia δ'. En esta definición las siguientes variables son siempre cuantificadas universalmente en sus respectivos dominios: u, v ∈ $(V^*)^{\leq n}$, w ∈ $(V^*)^{\leq |r|}$, μ ∈ $(V^* \cdot V_e^* \cup V_e^* \cdot V^*)^{\leq n}$, $\gamma \in \Gamma$, $x \in V$ and $Y \in V_e$.

$$\text{(i)}\ \delta'\left(\begin{bmatrix}\varepsilon\\ \varepsilon\\ xw\end{bmatrix}, x, \bot\right) = \left(\begin{bmatrix}\varepsilon\\ \varepsilon\\ w\end{bmatrix}, \bot\right) \text{ and } \delta'\left(\begin{bmatrix}\varepsilon\\ xu\\ xw\end{bmatrix}, \varepsilon, \bot\right) = \left(\begin{bmatrix}\varepsilon\\ u\\ w\end{bmatrix}, \bot\right)$$

$$\text{(ii)}\ \delta'\left(\begin{bmatrix}\mu xu\\ \varepsilon\\ w\end{bmatrix}, x, \gamma\right) = \left(\begin{bmatrix}\mu\langle x\rangle u\\ \varepsilon\\ w\end{bmatrix}, \gamma\right) \text{ and } \delta'\left(\begin{bmatrix}\mu xu\\ xv\\ w\end{bmatrix}, \varepsilon, \gamma\right) = \left(\begin{bmatrix}\mu\langle x\rangle u\\ v\\ w\end{bmatrix}, \gamma\right)$$

$$\text{(iii)}\ \delta'\left(\begin{bmatrix}u\langle x\rangle Y\mu\\ \varepsilon\\ w\end{bmatrix}, x, \gamma\right) = \left(\begin{bmatrix}uxY\mu\\ \varepsilon\\ w\end{bmatrix}, \gamma\right) \text{ and}$$

$$\delta'\left(\begin{bmatrix}u\langle x\rangle Y\mu\\ xv\\ w\end{bmatrix}, \varepsilon, \gamma\right) = \left(\begin{bmatrix}uxY\mu\\ v\\ w\end{bmatrix}, \gamma\right)$$

$$\text{(iv)}\ \delta'\left(\begin{bmatrix}u\langle x\rangle\\ \varepsilon\\ w\end{bmatrix}, x, \overline{\eta}\right) = \left(\begin{bmatrix}\eta\\ ux\\ w\end{bmatrix}, \varepsilon\right), \quad \delta'\left(\begin{bmatrix}u\langle x\rangle\\ xv\\ w\end{bmatrix}, \varepsilon, \overline{\eta}\right) = \left(\begin{bmatrix}\eta\\ uxv\\ w\end{bmatrix}, \varepsilon\right).$$

Fig. 11 - Función de transición intermedia

Todas las tripletas no listadas se igualan al conjunto vacío. Para obtener la función de transición δ de nuestro autómata, añadimos la posibilidad de ser interrumpido en cualquier punto de la computación de δ' y cambiarlo a un estado que comience la reducción de otra duplicación. Para ello definimos

$$\begin{bmatrix}\eta\\ v\\ w\end{bmatrix} \in Q \setminus F, \text{ and } \gamma \in \Gamma$$

$$\delta\left(\begin{bmatrix}\eta\\ v\\ w\end{bmatrix}, \varepsilon, \gamma\right) = \delta'\left(\begin{bmatrix}\eta\\ v\\ w\end{bmatrix}, \varepsilon, \gamma\right) \cup \left\{\left(\begin{bmatrix}z\\ v\\ w\end{bmatrix}, \overline{\eta}\gamma\right) \mid z \in (\Sigma^+)^{\leq n}\right\}.$$

Las transiciones (i) casan la palabra de la entrada y r, esto sólo se da con conjetura y pila vacías, que asegura que cada letra es mapeada sólo cuando todas las duplicaciones que a afectan han sido reducidas. Conjuntos (ii) y (iii) comprueban si la conjetura, que es el segmento de la supuesta duplicación, puede ocurrir dos veces adyacentemente en memoria seguidamente de la entrada. Esto se hace convirtiendo primero las letras de la conjetura en V a su correspondiente letra en V_e (conjunto ii) exactamente si su respectiva letra es leída de memoria o de la cadena de entrada. Entonces el conjunto (iii) recupera las letras originales. Finalmente, las transiciones en (iv) comprueban la última letra, si también coincide, entonces la duplicación es reducida dejando sólo una copia (de las dos leídas) de la conjetura en memoria, y la computación continúa volviendo a dejar la cadena de la cima de la pila de vuelta a la conjetura.

Hablemos de complejidad. La complejidad temporal de un autómata es cuántos movimientos tiene que realizar para discernir si acepta o rechaza la cadena de la entrada. En este caso la complejidad temporal será como mucho de $|x|^n$, siendo x la cadena de entrada y n la longitud máxima del segmento a duplicar. Puede no parecer mucho, pero recordemos que se trata de un autómata no determinista, por lo que si

intentamos implementar el autómata de pila tal cual en código tendríamos que crear un árbol de derivaciones que explotaría en número de posibles ramificaciones, generarlas todas y procesarlas todas para ver si alguna de ellas alcanza la aceptación de la entrada sería prohibitivo (tanto en memoria para mantener el árbol como en tiempo para procesar toras las ramas), así que he ahí nuestro problema, al cual le daremos solución a continuación.

En cuanto al número de producciones, tendremos una combinatoria bastante grande, algunos de los factores más importantes que disparan el número desorbitadamente son $w \in (V^*)^{\leq n}$ y $\mu \in (V^* \cdot V_e^* \cup V_e^* \cdot V^*)^{\leq n}$, siendo la de mayor relevancia la primera ya que las producciones son para toda w y esta son todas las cadenas de V* de longitud menor o igual a la cadena de entrada a partir de la cual montamos el autómata de pila. Tal número de producciones puede colapsar la memoria disponible.

4. Solución adoptada

Como hemos dicho, podríamos evaluar directamente las cadenas de test sobre el autómata de pila de aceptación por estado final que hemos definido anteriormente, pero al tratarse de un autómata no determinista se generaría un árbol de posibles computaciones que harían inviable su adaptación a un programa real, por lo tanto, enfocaremos la problemática en una solución en dos fases. La primera fase consistirá en obtener una gramática en Forma Normal de Chomsky (FNC) y la segunda será la evaluación de la gramática junto a la cadena que deseemos evaluar, haciendo uso del algoritmo Cocke-Younger-Kasami (CYK). Esto nos permite pasar de una ejecución no determinista a una determinista (ya que el proceso de ramificación se realiza una única vez cuando obtenemos la gramática y no con cada evaluación).

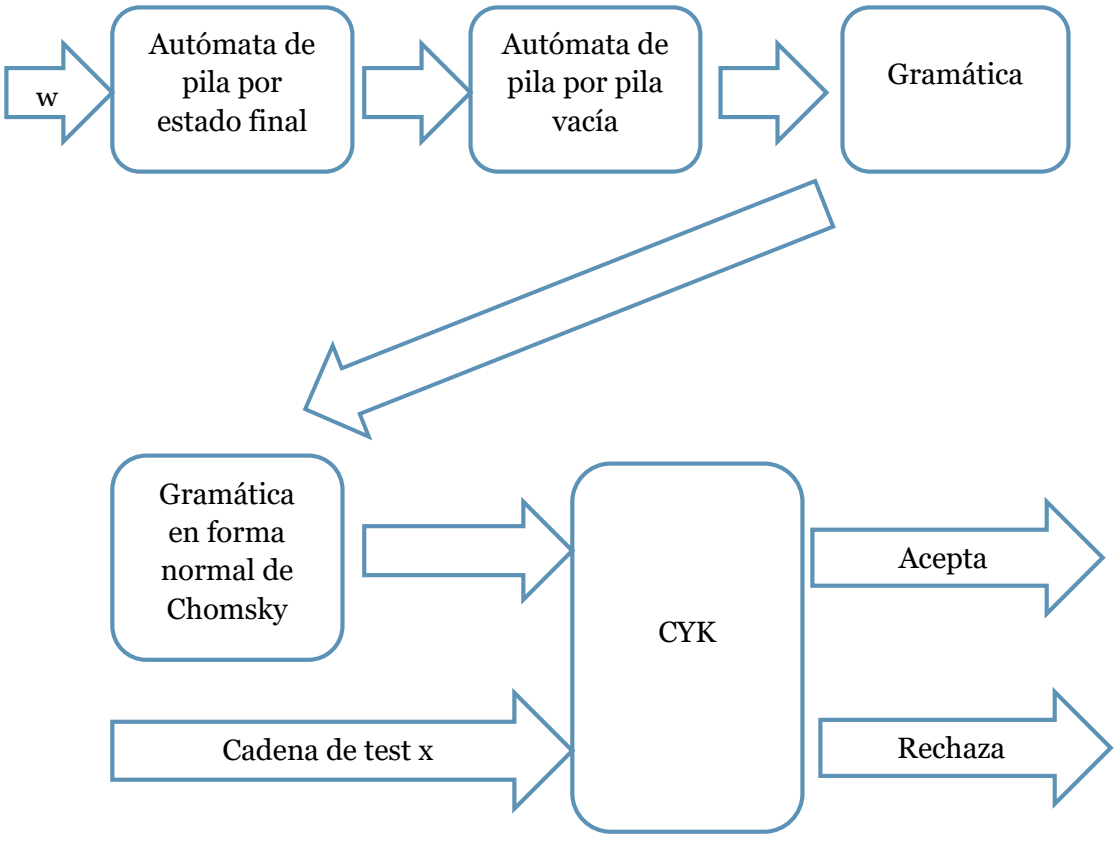

Fig. 12 - Esquema de resolución adoptado

Dada la cadena de entrada w, construimos primero el autómata de pila de aceptación por estado final que hemos descrito anteriormente $M_1 = (Q, \sum, \Gamma, \delta, q_o, Z_o, F)$. Seguidamente lo convertimos a uno de aceptación por pila vacía, para ello construimos un autómata como sigue:

$$M_2 = (Q \cup \{q_e, q'_o\}, \sum, \Gamma \cup \{X_o\}, \delta', q'_o, X_o, \emptyset)$$

donde δ’ se define de la siguiente forma.

1) $\delta'(q'_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$.
2) $\delta'(q, a, Z)$ incluye los elementos de $\delta(q, a, Z)$.
3) Para toda q en F, y Z en $\Gamma \cup \{X_0\}$, $\delta'(q, \varepsilon, Z)$ contiene (q_e, ε).
4) Para toda Z en $\Gamma \cup \{X_0\}$, $\delta'(q_e, \varepsilon, Z)$ contiene (q_e, ε).

Básicamente emula el autómata de pila de aceptación por estado final, primero coloca su propio símbolo inicial en la pila debajo del inicial de M_1, así en el caso de que la emulación vacíe la pila, no aceptará porque aún queda un símbolo en la pila. En el caso de que se alcance uno de los estados que antes era final, entrará en un modo de borrado continuo de la pila hasta dejarla vacía y haciendo que el autómata pare y acepte.

El siguiente paso es convertirlo a una gramática, partiendo del autómata de pila de aceptación por pila vacía M_2 anterior, que renombramos como $(Q, \sum, \Gamma, \delta, q_0, Z_0, \emptyset)$ para facilitar la lectura. Sea $G = (V, \sum, P, S)$ una gramática incontextual donde V es el conjunto de objetos de la forma [q, A, p], siendo q y p estados de Q y A perteneciente a Γ más el símbolo S.

Sea P el conjunto de producciones tales que

1) $S \rightarrow [q_0, Z_0, q]$ para cada q en Q;
2) $[q, A, q_{m+1}] \rightarrow a[q_1, B_1, q_2]\,[q_2, B_2, q_3]\ldots[q_m, B_m, q_{m+1}]$ para cada $q_1, q_2, \ldots, q_{m+1}$ perteneciente a Q, cada a perteneciente a $\sum \cup \{\varepsilon\}$, y A, $B_1, B_2, \ldots, B_m$ en Γ, tal que $\delta(q, a, A)$ contiene $(q_1, B_1B_2\ldots B_m)$. Si m = 0, la producción es de la forma $[q, A, q_1] \rightarrow a$.

Ya tenemos la gramática, pero todavía no es adecuada para el CYK, primero se eliminan las producciones unitarias de S, poniendo como consecuente de S los consecuentes de las producciones a las que apunta. Seguidamente se procede a eliminar las producciones vacías, aquellas que acaban transicionando a ε. Para ello el algoritmo tiene dos partes.

1) Obtener una lista de símbolos anulables, si $A \rightarrow \varepsilon$. A es anulable y se mete en el conjunto de anulables, si transiciona a una cadena, será anulable si y solo sí todos los símbolos de dicha cadena están en el conjunto de anulables.
2) Definimos la sustitución $g:(N \cup T)^* \rightarrow 2^{(N \cup T)^*}$ definida como
    1. Si $x \in (N \cup T) - \text{Anulables}$, $g(x) = \{x\}$
    2. Si $x \in \text{Anulables}$, $g(x) = \{x, \varepsilon\}$
    3. Si $z = xyz$, $g(z) = g(x)g(y)g(z)$
3) Definimos f(z) como $g(z)-\{\varepsilon\}$.
4) Construimos el nuevo conjunto de producciones $P' = \{A \rightarrow x: \exists(A \rightarrow z) \in P, x \in f(z)\}$

Para facilitar las cosas procedemos también por un renombrado de todos los símbolos auxiliares de la forma [q, A, p] a C_x. Ya tenemos la gramática sin producciones unitarias y sin producciones vacías, ahora podemos pasar a Forma Normal de Chomsky (FNC). El algoritmo consta de dos pasos:

PASO 1

N'=N; P'=∅;
Para toda regla ($A \rightarrow \alpha$) de P hacer
Si $|\alpha|=1$
entonces añadir la regla a P' (*ya esta en FNC*)
Sino sea $\alpha=X_1X_2...X_m$ con $m > 1$
Para i=1 hasta m hacer
Si $X_i=a \in \Sigma$
Entonces se añade a N' un nuevo no terminal C_a y se añade a P' una nueva regla ($C_a \rightarrow a$)
finsi
finpara
Se añade a P' una regla ($A \rightarrow X'_1X'_2...X'_m$)
con:
$X'_i=X_i$ si $X_i \in N$
$X'_i=C_a$ si $X_i = a \in \Sigma$
finSi
finPara

Fig. 13 - Primera parte del algoritmo de paso a FNC

PASO 2

(*Se toma como entrada la gramática G' resultante del PASO 1*)

N''=N'; P''=∅;
Para toda regla ($A \rightarrow \alpha$) de P' hacer
Si $|\alpha| < 3$
Entonces añadir la regla a P'' (*ya esta en FNC*)
Sino sea $\alpha = B_1B_2 ...B_m$ con $m > 2$
Añadir a N' los no terminales { $D_1, D_2, ..., D_{m-2}$};
Añadir a P'' el siguiente conjunto de reglas:
$A \rightarrow B_1D_1$
$D_1 \rightarrow B_2D_2$
...
$D_{m-3} \rightarrow B_{m-2}D_{m-2}$
$D_{m-2} \rightarrow B_{m-1}D_m$;
finSi
finPara

La gramática resultado es G''=(N'',T,P'',S).

Fig. 14 - Segunda parte del algoritmo de paso a FNC

El algoritmo Cocke-Younger-Kasami permite evaluar si una cadena de test pertenece a la gramática incontextual que se le pase en FNC, sacando por la salida si dicha cadena pertenece a la gramática o no. El coste temporal del algoritmo es cúbico con la longitud de la cadena, lo que lo hace muy eficiente comparado con otros algoritmos. La estructura del algoritmo es la que sigue Con G = (N, T, P, S) en Forma Normal de Chomsky y una cadena de entrada $w = w_1w_2...w_n$ y w distinto de ε.:

```
Para i=1 hasta n
    Vi1 = { A : A → wi ∈ P }
finPara
Para j=2 hasta n
   Para i=1 hasta n-j+1
        Vij = ∅
        Para k=1 hasta j-1
            Vij = Vij ∪ { A : A → BC ∈ P, B ∈ Vik, C ∈ Vi+k,j-k }
        finPara
    finPara
finPara
Si S ∈ V1n devolver Cierto sino devolver Falso
```

Fig. 15 - Algoritmo CYK

Veamos un ejemplo de su funcionamiento:

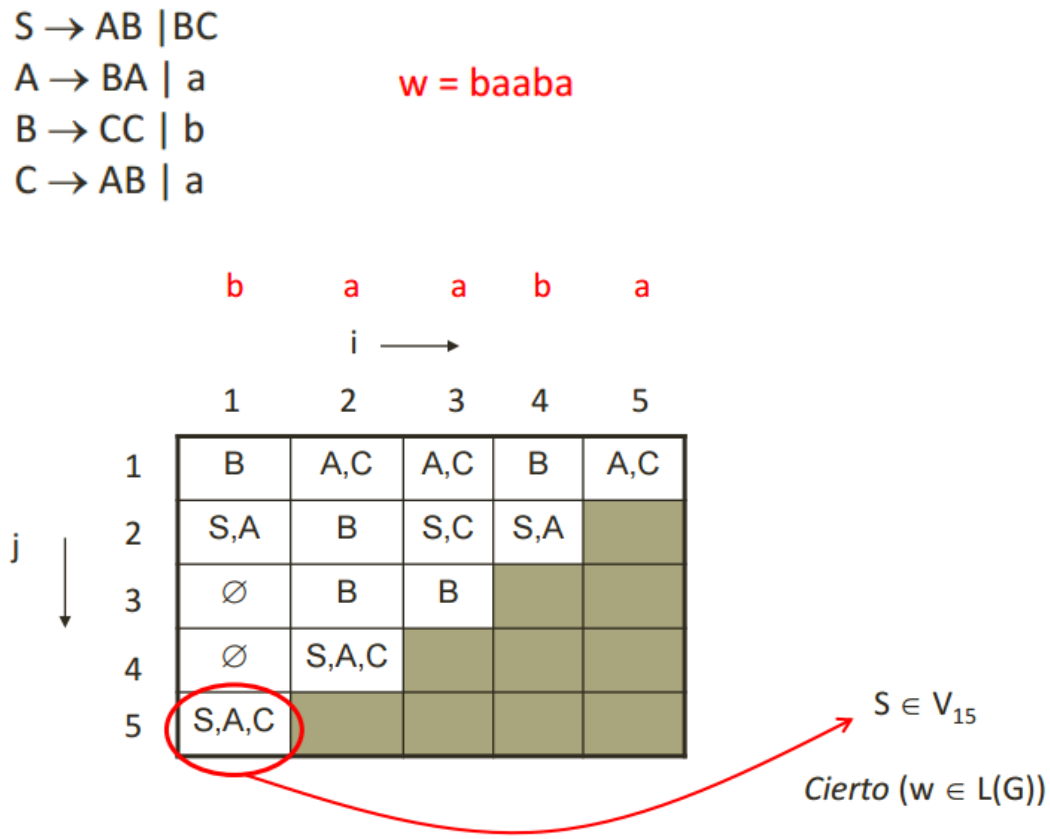

Fig. 16 - Ejemplo de ejecución CYK

En cada celda de la tabla V_{ij} denota los símbolos auxiliares que pueden derivar, en *G*, el segmento de la cadena de análisis x, que comienza en la posición i y tiene longitud j. Se trata de ir haciendo todas las posibles reducciones con todas las celdas del rango y guardando los antecedentes.

En la última celda (V_{1n}) están los posibles símbolos desde los cuáles se ha podido generar la cadena de entrada, si uno de ellos es el símbolo inicial de la gramática, quiere decir que la cadena w pertenece al lenguaje *L* generado por la gramática *G*.

Por lo tanto, si la cadena de test que queremos probar es aceptada por la gramática que hemos generado previamente, quiere decir que se trata de una posible mutación por duplicación de la cadena original.

5. Implementación de una herramienta

Para realizar la implementación de los algoritmos que se han descrito anteriormente, se decidió aceptar la sugerencia hecha por el director del proyecto, el cual requería que el analizador se implementara en lenguaje JAVA, debido principalmente a la posible incorporación en un futuro, de la herramienta de análisis como aplicación web.

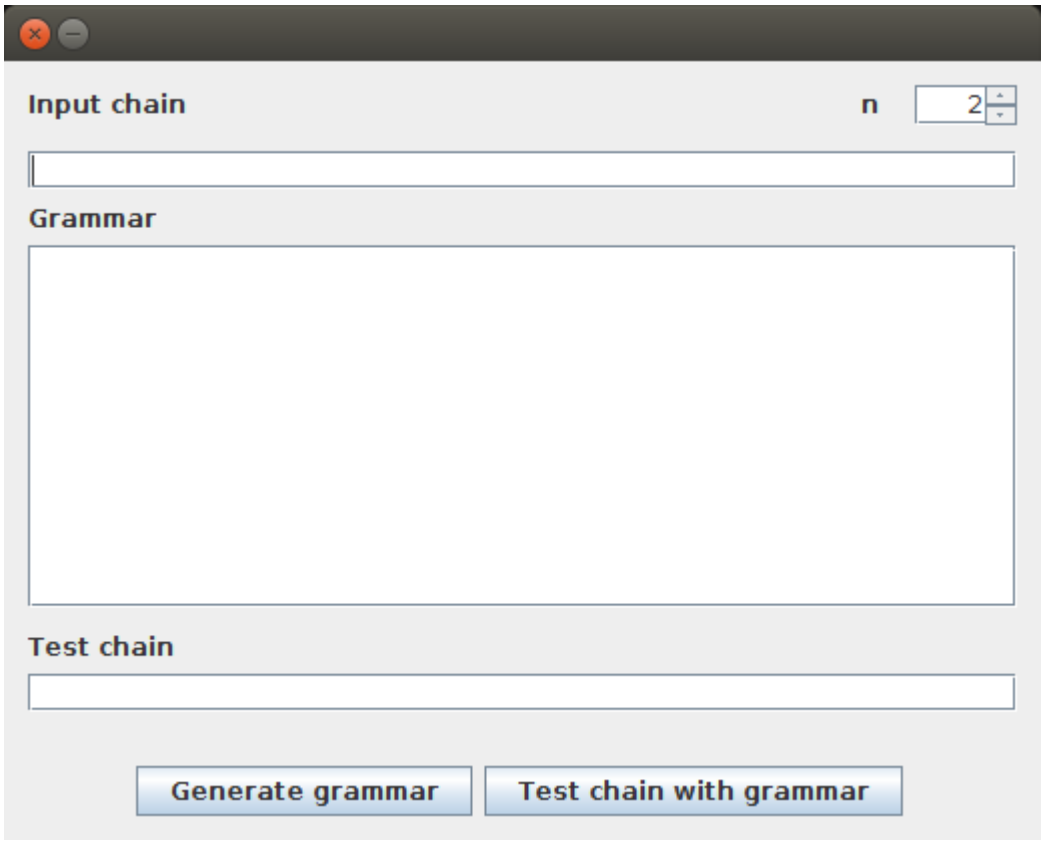

Fig. 17 - Captura de pantalla de la herramienta

La interfaz de usuario está compuesta por tres campos de texto, un campo numérico y dos botones de función, cada uno de los cuales lleva asociada la correspondiente etiqueta que lo define.

El primer campo es para la cadena de entrada a partir de la cual se genera la gramática de duplicación y está acotada a duplicidades de tamaño n (el selector de arriba a la derecha). Al pinchar sobre “Generate grammar” se pone en marcha la primera fase de la solución adoptada y como resultado de esta se escribe en el cuadro de texto etiquetado como “Grammar”. Debido al alto coste computacional, parte del código está paralelizado para aprovechar las capacidades de los procesadores con

múltiples hilos de ejecución (ejecuta tantos hilos de JAVA como el número de hilos simultáneos que puede ejecutar el procesador).

El segundo botón toma la gramática del cuadro de texto y la cadena de test como entrada del algoritmo CYK, entrando en juego la segunda fase de la solución adoptada. Esto es así para permitir al usuario que guarde la gramática resultante de una ejecución (seleccionando y copiando del cuadro de texto) para posibles pruebas futuras y que pueda introducir una gramática previamente calculada.

Al finalizar la ejecución se muestra por pantalla una ventana emergente que dice si la cadena de test es aceptada por la gramática computada previamente o si es rechazada. Finalmente, si el usuario quiere probar otras cadenas de test, sólo tiene que borrar la introducida y poner la nueva, todo esto sin tocar la gramática y sin tener que volver a generarla.

A la hora de programar la herramienta se ha seguido un esquema de segmentos independientes donde cada apartado de cada fase es una "caja" con unas entradas y una salida, totalmente independientes las unas de las otras, así en caso de fallar alguna o de tener que cambiar algo de su interior no afecta a las demás cajas (salvo que se toque la salida o la entrada).

Además, se han codificado las "cajas" de atrás hacia adelante, es decir, empezando primero por la última del esquema (el algoritmo CYK) e ir haciendo aquellas que son requisito previo de la que se acaba de codificar.

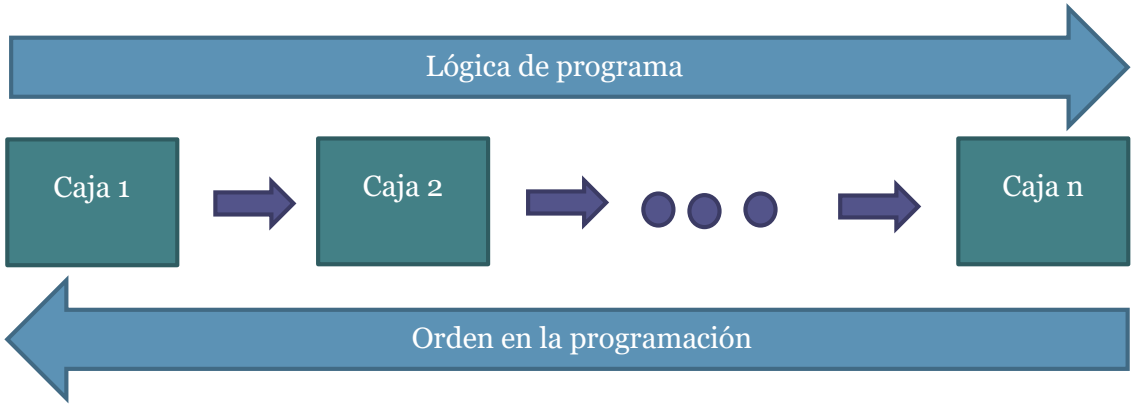

Fig. 18 - Esquema de la programación

Esto permite ir programando cada una de las cajas sabiendo qué es lo que tiene que aportar a la siguiente caja, proporcionando en la salida la información justa, ni más ni menos, así como evitando tener que implementar aspectos de los algoritmos que consumen mucho tiempo de cálculo y memoria si luego no se van a utilizar.

El primer problema a afrontar ha sido cómo representar en JAVA los elementos del algoritmo, los símbolos, las producciones de la gramática, la gramática en sí, etc... Se optó en primera instancia en que cada símbolo fuese un objeto, y por lo tanto una cadena de símbolos fuese una lista de objetos, pero posteriormente esto se tuvo que

cambiar por cadenas de caracteres separados por comas porque se generaba demasiada información irrelevante en memoria.

En segundo lugar, otro problema afrontado ha sido la falta de uso reciente del lenguaje, debido a las múltiples asignaturas que utilizan otros lenguajes de programación, el dominio de JAVA que pudiese tener se ha visto mermado y ha sido como enfrentarse casi desde cero a un nuevo lenguaje.

Otro de los problemas ha sido el tener que trabajar con listas anidadas, una tarea fácil y cómoda en lenguajes como Python o Mathematica, pero que en JAVA es una tarea nada agradable de programar, acabando con código nada legible e inmanejable.

El proceso de generación de la gramática es muy exigente en cuanto a cantidad de memoria necesaria (hay que recordar que se generan todas las posibles combinaciones de movimientos que puede realizar el autómata de pila original), y unido a la pobre gestión de la memoria que hace JAVA hace que la ejecución termine siempre con un error de memoria insuficiente (aunque se hayan hecho pruebas con 22GB de tamaño máximo de *heap*) o por tiempo excedido del *garbage collector* (cuando se pasa el 98% de tiempo de ejecución intentando liberar memoria y no lo consigue).

Por consiguiente, una de las posibles soluciones que se podría plantear al tema de la memoria (aparte de utilizar otro lenguaje con mejor uso de la memoria) es la de ejecutarlo en un clúster de supercomputación con grandes cantidades de memoria, o también, en la nube, donde se puede ir contratando y añadiendo más memoria según va necesitando el programa.

6. Conclusiones

El presente trabajo ha sido un auténtico reto, tanto a nivel personal como profesional. Esta ha sido la primera vez que he realizado programación de alto nivel con un proyecto en el que he implementado más de dos mil líneas de código. Durante su realización me he encontrado con varios problemas, que paso a relatar.

En primer lugar, me he visto limitado por la poca versatilidad del lenguaje JAVA, en cuanto al uso de listas anidadas se refiere (*ArrayList*), puesto que es el único objeto que he podido encontrar que se ajustara a los requerimientos del analizador.

En segundo lugar, y no por ello menor en importancia, me encontré con el problema de la gestión de la memoria…. No he conseguido confirmar si los problemas de memoria son sólo los ocasionados por el *garbage collector*, o si también han influido y en qué medida, las características físicas de mi ordenador.

Uno de los principales problemas ha sido la gestión de memoria (tal y como se mencionó en el apartado anterior). Intentar una representación de alto nivel de la solución adoptada mediante objetos para representar símbolos y demás elementos, ocasionó que mucha memoria disponible se gastase en guardar información relacionada con los objetos, en lugar de guardar información útil. Los movimientos del autómata se codifican como un *ArrayList* de objetos, que a su vez cada objeto contiene un *ArrayList* de símbolos. Las producciones de la gramática son una lista de objetos de tipo *production* que a su vez contienen una lista de los posibles consecuentes, siendo cada entrada una lista de símbolos.

Para solucionar este problema decidí realizar una reescritura del programa en prácticamente su totalidad, enfocando la programación del mismo a más bajo nivel (utilizar representación en cadenas, montando estructuras separadas por comas como serían los distintos atributos de un objeto). Pero esto sólo solucionó el problema de memoria temporalmente, volviendo a resurgir en un tiempo más avanzado de ejecución, dada que la información que se genera es tal que satura la memoria.

Queda como sugerencia para un futuro proyecto, la realización del analizador en otro lenguaje de programación, que permita una gestión de memoria más eficaz y con mayor sencillez en la representación algorítmica; como por ejemplo podría ser Python, en el que el *garbage collector* es más sencillo, pero más eficaz, también cabe resaltar que el consumo de tiempo de ejecución suele ser mucho inferior que en JAVA.

Hablo de Python y no de C o C++, porque, aunque son mucho más eficientes en la gestión de la memoria y son más fáciles de paralelizar gracias a las directivas del precompilador para usar OpenMP y MPI, la implementación en código del algoritmo sería mucho más costosa de realizar al tratarse de lenguajes de más bajo nivel. Sugiero Python porque es una solución intermedia en cuanto a eficiencia (corre sobre C) y porque es uno de los lenguajes punteros en el ámbito de la computación de altas prestaciones (minería de datos, por ejemplo), siendo con diferencia el lenguaje más usado en matemáticas y datos según StackOverflow.

Para finalizar, me siento satisfecho con el trabajo realizado, dado que he empleado muchas horas de dedicación en exclusiva a este proyecto, que me ha ilusionado y motivado desde el primer momento hasta el final. Los problemas que me he encontrado durante su realización me han llevado a pensar que sería muy interesante poder ampliar la visión del proyecto, como he mencionado antes, realizando una comparativa de lenguajes de programación, puesto que el coste temporal y espacial en este caso son sumamente importantes, y habría que gestionarlos de la forma más optimizada posible. Me reservo el privilegio, a modo de reto personal, de realizar la herramienta en Python.

De cara a la herramienta, posibles mejoras podrían ser, por ejemplo, que en el caso de aceptación explicase dónde y por qué se ha dado esa duplicación, o la integración en web e interacción con otras herramientas similares que traten sobre otras mutaciones.

7. Índice de figuras

8. Bibliografía

[1] Alberts, J., Lewis, R., Roberts, W. (2008) Biología Molecular de la Célula Ediciones Omega. 62

[2] Leupold, P., Mitrana, V., Sempere, J. M. (2004) Formal Languages Arising from Gene Repeated Duplication. 299-306

[3] Karhumäki, J. (1980) Generalized Parikh mappings and homomorphisms, Elsevier. 47:155-165

[4] Dassow, J., Mitrana, V., Paun, G. (1999) On the regularity of duplication closure, Bull. EATCS, 69:133-136

[5] Ming-wei, W. (2000) On the irregularity of the duplication closure, Bull. EATCS, 70:162-163

[6] Hopcroft, J. E., Ullman, J. D. (1979) Introducing to Automata Theory, Languages, and Computation, Addison-Wesley, 108-110